TASK: PA18
CDRL: A023
26 February 1994

Methodology For Verifying Ada Tasking With Penelope

Informal Technical Data

STARS-AC-A023/003/00
26 February 1994

TASK: PA18
CDRL: A023
26 February 1994

INFORMAL TECHNICAL REPORT
For
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Methodology For Verifying Ada Tasking With Penelope

STARS-AC-A023/003/00
26 February 1994

Data Type: Informal Technical Data

CONTRACT NO. F19628-93-C-0130

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

Odyssey Research Associates
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 26 Feb 1994	3. REPORT TYPE AND DATES COVERED Informal Technical Report
4. TITLE AND SUBTITLE Methodology For Verifying Ada Tasking With Penelope		5. FUNDING NUMBERS F19628-93-C-0130
6. AUTHOR(S) ORA		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Unisys Corporation 12010 Sunrise Valley Drive Reston, VA 22091		8. PERFORMING ORGANIZATION REPORT NUMBER STARS-AC-A203/003/00
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Air Force Headquarters ESC Hanscom, AFB, MA 01731-5000		10. SPONSORING/MONITORING AGENCY REPORT NUMBER A023
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution "A"		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper sketches a method for Penelope to support verification of Ada programs that use tasking.

14. SUBJECT TERMS			15. NUMBER OF PAGES 49
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

TASK: PA18
CDRL: A023
26 February 1994

Data Reference: STARS-AC-A023/003/00
INFORMAL TECHNICAL REPORT
Methodology For Verifying
Ada Tasking With Penelope

TASK: PA18
CDRL: A023
26 February 1994

Data Reference: STARS-AC-A023/003/00
INFORMAL TECHNICAL REPORT
Methodology For Verifying
Ada Tasking With Penelope

Principal Author(s):

Douglas N. Hoover — *Date*

Approvals:

[illegible] — 2/28/94

Program Manager *Teri F. Payton* — *Date*

(Signatures on File)

Contents

1 Introduction and Aims

This paper sketches a method for Penelope to support verification of Ada programs that use tasking. In developing this method, our aims have been the following.

1. The method must be compatible with Penelope. That is, program annotations should be written in classical logic, and Penelope should generate verification conditions whose proof verifies the program.
2. Program annotations should contribute to the understanding of the program by stating properties that must hold at that point of the computation.
3. The verification process must be compositional—that is, we must be able to verify parts of a program separately and then put them together to obtain a verification of the whole without repeating the verification of the parts.
4. The compositionality should be syntax directed—that is, the verification must break up along the lines of Ada tasks, subprograms, and packages.
5. The method should support separation of concerns in the verification process. For example, it should be possible to separately verify partial correctness, freedom from deadlock, and liveness without too much duplication of labor.
6. The method should support both data refinement and program refinement. Whenever possible, verification of properties expressible at more abstract levels should be inherited by the implementation.
7. If no tasks or global variables shared by tasks are visible from a given subprogram or package, then verification in that subprogram or package should be carried on in the sequential style, even if there are tasks contained inside visible subprograms or packages.

Properties of concurrent programs can be partitioned in two ways:

- local vs. inter-task properties; and
- safety vs. liveness properties.

Local properties are those that can be regarded as properties of an individual subprogram or task. Inter-task properties are those that describe the interactions of a group of tasks. Safety properties state that the program will always be in an acceptable state. Liveness properties state that some desired action will eventually be performed.

Local safety properties are just partial correctness properties: we must show that each annotation in a task or subprogram holds whenever it is reached. Local liveness properties are similar to proving termination of a sequential program. For example, one may want to prove that the execution of an individual task or subprogram will reach completion provided all the task entry or subprogram calls it makes terminate, or one may wish to show that execution of certain segments of a task terminate, in case the task as a whole is not meant ever to reach completion on its own.

The most important inter-task safety property is deadlock freedom. That is, it is never the case that all active tasks are blocked waiting to communicate with other tasks that are blocked elsewhere. A typical inter-task liveness property is to show that some chain of communications between tasks will eventually be completed.

In general, it appears that partial correctness is the most important thing. Local liveness properties are proved by proving partial correctness with appropriate annotations, and proving global properties seems to be mainly a matter of organizing local properties in the right way.

For proving partial correctness properties, we present a method based on that of Owicki-Gries [10]. For proving global properties, we outline a state machine method, in which the states are states of tasks at synchronization points. That a given state machine models a system of tasks is to be derived by partial correctness methods. We do not treat liveness properties (total correctness) in detail here, but we discuss it briefly in Section 15.

Our method for specifying and verifying partial correctness is based on the Owicki-Gries method [10], a natural extension of Hoare logic to concurrent programs. Since Penelope is based on predicate transformation, which is essentially a partially automated Hoare logic, Penelope's extension to Ada tasking is naturally based on an adaptation of the Owicki-Gries method.

Besides adapting Owicki-Gries to the predicate transformation format, we have adapted it in two other ways. In the original Owicki-Gries paper, processes communicate only via shared variables. We have adapted the method to support typical Ada communication via rendezvous (task entry calls—the Owicki-Gries method was also adapted for this purpose in [3]). The Ada method of communication is much cleaner and has the attractive feature that the semantics of task entry calls resemble those of subprogram calls.

The second way in which we have modified the Owicki-Gries method is to support composable, reusable verification. The problem with Owicki-Gries is that

- verification of each task involves a global cooperation invariant (called a *global invariant* by Owicki and Gries);
- what the cooperation invariant will be depends on the whole ensemble of tasks and their global pre- and post-conditions;
- therefore no task can be verified without knowledge of the whole ensemble of tasks with which it is to be used.

We get around this problem by observing that the verification of each task does not depend very strongly on the cooperation invariant (CI). Rather, it requires only that the cooperation invariant satisfy certain properties. We express these properties as a constraint on the CI. The CI constraint for a collection of tasks is just the conjunction of the constraints on the individual tasks. The cooperation invariant itself need not be known to the individual tasks, only to their parent.

In order to eliminate certain problems in dealing with cooperation invariants, to permit more intuitive specification of entry calls, and to facilitate program refinement, we have made free use of virtual variables. By using virtual variables where necessary, task entry semantics can be made very similar to subprogram semantics.

Our use of virtual variables has been inspired by Lamport's work on refinement [8]. Lamport emphasizes that the abstraction relation between the abstract state and the implementation state need not hold at all times. Our virtual variables stand for elements of an abstract state that need coincide with the desired abstraction of the implementation state only when the specifier says that they must.

1.1 Relation to Other Work

Cooperation invariant constraints and use of virtual variables to make entry call semantics analogous to subprogram call semantics are the main innovations of our approach. We feel that they constitute a significant improvement on previous attempts to give Ada tasking semantics [3, 2].

Owicki and Gries [10] are concerned with communication via shared variables. In that context, the cooperation invariant will be a relation that must (almost) always hold among the shared variables. But assignments to shared variables can occur only one at a time. Thus, if a group of assignments is to be made to several global variables, the cooperation invariant will probably not hold between the first assignment and the last. In order to handle this problem, Owicki and Gries adopt a method of bracketing critical regions in which the cooperation invariant need not hold, so that groups of assignments to shared variables can be made. Only one task may be in a critical region at a given time. The cooperation invariant must be preserved by each critical region treated as an atomic unit.

In Ada, process communication is normally by entry call rather than by shared variables, and the cooperation invariant normally relates non-shared variables that constitute the states of the various tasks. The problem is that the natural cooperation invariant tends to be violated when communications occur.

For example consider two tasks $T1$ and $T2$ that manage two buffers $B1$ and $B2$. Suppose that the object is to pass the contents of $B1$ to $B2$ a character at a time. The cooperation invariant will be

```
B1 ++ B2 = IN B1 ++ IN B2
```

that is, the current contents of $B1$ and $B2$ are the same as their original contents. A communication will consist of $T1$ sending $T2$ the last character c in $B1$, which $T2$ will add to the front of $B2$.

Now $T1$ will probably remove c from $B1$ before the communication commences, and $T2$ will add it to $B2$ only after the communcation completes, so that the cooperation invariant is violated at the time of the communication. Gerth and de Roever [3] solve this problem by a bracketing method similar to the critical regions of Owicki and Gries. Each communication is to be bracketed in both of the tasks involved, and the CI is not required to hold in the bracketed region. (Dillon [2] extends this method slightly to reduce the use of virtual variables.)

Unfortunately this approach is rather messy. Proving the correctness of each communication requires knowing code from the bracketed regions in both the communicating tasks, making it impossible to verify tasks in isolation, and making the whole method non-compositional. Even worse, the possibility of nested accept statements and entry calls inside accept bodies in Ada means that a more complicated split bracketing sometimes has to be used.

We have solved this problem by using virtual variables. In the example, virtual variables $B1'$ and $B2'$ are used in the cooperation invariant. The communication is considered to instantaneously transfer a character from $B1'$ to $B2'$, thereby preserving the cooperation invariant. As part of the verification we will have to show that $B1' = B1$ and $B2' = B2$ at suitable times, but that can be done in isolation for each task. With our approach, the semantics of a task entry call is similar to the semantics of a subprogram call, except that we must show that the cooperation invariant is preserved.

Jones [6, 7] suggested another method for giving a compositional extension of Hoare to concurrent programs. We have preferred the Owicki-Gries method on the grounds that it is more comprehensible. It is not always obvious what the annotations should be in Jones's method.

Jones required that each task be annotated with the following conditions:

- an invariant that the task assumes other tasks preserve, but that the task need not preserve;
- an invariant that the task would guarantee if it were running in isolation, but that other tasks need not preserve;
- a precondition;
- a postcondition.

Jones gave a verification rule due to P. Aczel for parallel composition of tasks with these annotations, yielding verification conditions relating the annotations of the composed tasks to their composition.

We feel that Jones's method is rather confusing for the following reasons. First, the condition that a task guarantees is not only a condition that it would preserve if it were running in isolation; other tasks may violate it. This fact makes it hard to figure out what the guarantee should be, since it is not something that will actually be an invariant in the whole system. Furthermore, what would intuitively be the postcondition—what we want the task to establish—has to be teased apart in a rather delicate way, making part of it into the Jones postcondition and part into the Jones guarantee. It was not obvious how to do this for the buffer example, even though it is closely related to the distributed sieve of Eratosthenes example that Jones works in [6] (and we work in Appendix B). Since, with a given purpose one has in mind, it is usually quite obvious what the Owicki-Gries cooperation invariant should be, and one can derive from the verification process itself what constraints a given task imposes on the cooperation condition, it seems easier to use the Owicki-Gries method.

1.2 Overview

The proof of total correctness of a tasking program can be divided into three parts, two of which are essentially the same as the two parts (partial correctness and termination) of the proof of total correctness of a sequential program:

- partial correctness, the proof that each embedded assertion is true whenever it is reached;
- freedom from deadlock, the proof that computation does not hang up because some process is waiting for a rendezvous that is never kept;
- progress (termination, freedom from livelock), the proof that computation does not just spin its wheels, but produces desired results.

We feel that in constructing a verification system for tasking programs, these three aspects should be addressed in this order. One reason for taking this point of view is that a proof of partial correctness is the most interesting semantically. The other reason is that deadlock freedom and progress conditions are proved by embedding particular annotations in the program and giving a partial correctness proof that they are true whenever they are reached. Consequently, we concentrate most on partial correctness in this report, though we also cover the other aspects of verification.

We will begin our discussion of semantics by sketching the essential idea of the Owicki-Gries method in Section 2. In Sections 3 and 4, we will develop its application to Ada tasking semantics using the eternal producer-consumer (buffer) example. In Section 5 we develop the method of cooperation invariant constraints as a way to make the Owicki-Gries method compositional. In Section 7 we discuss the semantics of the `delay` statement. In Section 8 we introduce a language for specifying control flow of concurrent programs. This control flow language is exploited in Sections 9, 10 and 11 where we discuss exceptions, the abort statement, shared variables, and proofs of deadlock freedom. In Section 12 we discuss the semantics of tasks as formal parameters to subprograms. In Sections 13 and 14 we discuss an alternate style of verification and the distinction between analysis of control flow and of data transformation in concurrent programs. In Section 15 we introduce a way of modeling Ada tasking programs as state machines and discuss how to use Penelope to prove general temporal properties of concurrent programs. In Appendix A we summarize the semantics of Ada tasking primitives. In Appendices B and C we give more interesting examples of tasking programs, a distributed sieve of Eratosthenes and a systolic matrix multiplication algorithm. An example similar to the latter was worked in [12, 13]. The reader should compare the two approaches.

In this report we give semantics in Dijkstra's weakest precondition notation. Compared to the semantics in the Penelope predicate transformer document [11], this semantics is simplified by ignoring the environment, exceptions, and the possibility of side effects on expression evaluation, and thereby gains in understandability. Generalizing these semantics to a proper predicate transformer semantics is laborious but straightforward.

2 Concurrent Programming and the Owicki-Gries Method

Concurrent programming means that a family of program tasks is running "at the same time." "At the same time" might mean that their statements are really executed simultaneously on a collection of processors, or that they are executed in an interleaved fashion by time-sharing. Just how it is done does not affect the result as long as two tasks are not trying to access the same shared variable in statements whose order of execution is not determined. For most of this report, we will assume that there are no truly shared variables. We will allow tasks to use global variables, as long as there is only one active task at any one time that uses a given global variable. In Section 10 we will consider the complications introduced by shared variables.

We can write the composite of n tasks $T_1, \ldots, T_n$ running concurrently as

$$T_1 \parallel \ldots \parallel T_n,$$

(N.B.: This notation is only in this section for explanation. It will not be used in Penelope verification.)

Suppose that for some properties P and Q we want to show

$$\{P\}\ (T_1 \parallel \ldots \parallel T_n)\ \{Q\}. \tag{1}$$

(This Hoare triple notation says that the precondition P ensures that the postcondition Q holds after executing $T_1 \parallel \ldots \parallel T_n$.) If there are no communications between the different tasks, then the proof of this assertion reduces to finding conditions P_i and Q_i, $i = 1, \ldots, n$, such that for each i,

$$\{P_i\}\, T_i\, \{Q_i\},$$

as well as

$$\vdash P \rightarrow \bigwedge_i P_i$$

and

$$\vdash \bigwedge_i Q_i \rightarrow Q.$$

Normally, however, the different tasks will communicate, providing both a responsibility and an opportunity to show that

1. the interference between tasks does not spoil the verifications; and that
2. the cooperation between tasks leads to an accomplishment greater than they would have achieved separately.

This combination of responsibility and opportunity is met by introducing the *cooperation invariant* (`CI`). To prove (1) it suffices to show the following.

1. `CI` is preserved by all communications and internal computations of each task (i.e. it really is an invariant).
2. $\vdash P \rightarrow \mathtt{CI} \wedge \bigwedge_i P_i$.
3. $\vdash \mathtt{CI} \wedge \bigwedge_i Q_i \rightarrow Q$.

How we prove that `CI` is really an invariant will emerge as we work out a verification example in the following sections. We want to note here, though, that the use of a cooperation invariant poses a problem for composability of verifications. The reason is that, on the one hand, the choice of cooperation invariant depends on the whole ensemble of tasks that are cooperating, while, on the other hand, the verification of an individual task is not complete without the proof that the cooperation invariant is preserved by that task and its communications. In Section 5 we solve this problem by giving an isolated task a generic invariant together with a set of constraints that must be satisfied when the generic invariant is eventually instantiated by a concrete invariant.

3 Task and Task Entry Specifications

The tasks of an Ada program at a given moment consist of the main thread of control (the "sequential part") and whatever Ada tasks are currently active. The Ada program is the parallel composition of all these tasks. Although Ada tasks are pieces of program, Ada considers them data objects (though with no assignment or equality test) in order to accommodate arrays and lists of tasks and dynamic creation of tasks.

Communication between tasks is normally implemented using *entry* calls, as declared in the following declaration of a buffer task.

```
task BUFFER is
  entry write(c:  character --:  virtual stream_in:  in out Stream
    );
  entry read(c:  OUT character --:  virtual stream_out:  in out Stream;
    );
end BUFFER;
```

The annotation

```
--:  virtual
```

indicates that what follows are virtual parameters to the entry call. Semantically, they are like concrete parameters, but they have a sort rather than a type.

From the point of view of the caller, task entries look like procedure calls. Ada encourages this point of view by permitting entry calls to be renamed as if they were subprograms. There are, however, differences between a subprogram and a task entry.

1. A task can continue computing when none of its entries are being called.
2. A task can have several entries of the same name.
3. A task entry can be called only when the control point of the task is at the beginning of the entry. (Hence a task cannot call any of its own entries.)
4. Only one task at a time can rendezvous with a given task entry.

The first two items are internal considerations for the task containing the entry. The only external effect of the latter two items is the possibility of deadlock. From the point of view of partial correctness, therefore, it is reasonable to treat a task entry externally as if it were a procedure.

We illustrate this with the following example of a system consisting of three tasks, *PRODUCER*, *BUFFER*, and *CONSUMER*.

1. The task *PRODUCER* sends characters from an array A to a queue.
2. The task *BUFFER* manages this queue.
3. The task *CONSUMER* takes characters from the queue and puts them in the array B.
4. When all the characters have been moved, the program returns.

This program simply copies an array and is used for illustration. We start with the package specification.

```
package PBC is
  --| globals: A : in, B: out;
  --| PRE: A'length = B'length and A'length > 0;
  --| POST B = in A;
end PBC;
```

The globals annotation must list all global variables that may be read or written by tasks inside the package. Consider an annotation:

```
        --| globals V : in;
```

This annotation means that the global V may be read (but not written) by tasks inside the package. This annotation invokes machinery to enforce the mutual exclusion conditions that the Ada Reference manual (RM) Section 9.11 imposes on use of global variables by tasks, but as long as each global variable is declared as a global by only one task, we do not have to worry about this machinery.

The PRE and POST annotations indicate the conditions that PBC assumes at the beginning of its activation (RM 9.3) and that it guarantees will hold when it and its dependents are terminated or ready to terminate (RM 9.4). We will see how they are used in the next section.

Note that the declaration of PBC does not contain any task or subprogram declarations. That is because when the body of PBC is elaborated, its tasks will be set going automatically and will do their work without any outside prompting. If task entries inside PBC were externally visible, that would not materially complicate proof of partial correctness.

```
package body PBC is

  task PRODUCER;
        --| STATE to_Q : Stream := [];
        --| PRE: true;
        --| POST: to_Q = A;

  task BUFFER
        --| STATE Q : Stream := [];
        --| PRE: true;
        --| POST: Q = [];
       is
        entry WRITE(c: character --: virtual stream_in: in out Stream
                    );
                --| OUT Q = [c] ++ (IN Q);
                --| OUT stream_in = c ++ (IN stream_in);
        entry READ(c: OUT character --: virtual stream_out: in out Stream
                    );
                --| PRE Q /= [];
                --| OUT (IN Q) = Q ++ [c] ;
                --| OUT stream_out = [c] ++ (IN stream_out);
  end BUFFER;

  task CONSUMER;
     --| STATE to_Q : Stream := [];
     --| OUT from_Q = B;

  --| CI: PRODUCER.to_Q = BUFFER.Q ++ CONSUMER.from_Q;

end PBC;
```

Stream is the sort of lists of characters. The STATE of a task (or, more generally, of any declarative region) is a list of virtual and actual variables. The actual variables are to be declared later in the body of the task. The state declaration will suffice as a declaration of the virtual variables. Initial values may be supplied for the variables in the state. Actual variables in the state must get the same initialization when they are eventually declared as actual variables. The CI clause declares the cooperation invariant.

State variables are a restricted form of global variables. They may appear in the CI, but not in any other code or annotations outside the task that declares them. (They will occur outside the task that declares them in verification conditions (VCs) connected with the CI.) The purpose of the state is to make private variables of the task available to the CI. Global variables may not be included in the state.

Note that the task entries can have virtual formal parameters. The corresponding actual parameters will normally be state variables of the task that calls the entry.

A task entry may have IN, OUT, and PRE conditions. Proof obligations are distributed as follows.

- The verification of the calling task must prove that the CI and the IN condition hold whenever the entry is called.
- The verification of the accepting task must prove that the CI and the PRE condition hold whenever the accept statement is reached.
- Verification of the accept body must show that the OUT condition holds whenever the accept statement returns.
- The calling task must show that the CI and the postcondition of the task both hold when the entry call returns. In doing so, it may use the facts that the PRE condition and the CI held before the entry call.

- The postcondition of the accept statement can just be transformed through the accept statement.

Except for the verification of the individual tasks and the proof that they preserve the CI, we can now derive and prove all the VCs related to this ensemble of tasks. The VCs are as follows, written in sequent form.

- PRE condition of PBC implies the CI with the initial values of the state variables of its components. In sequent form,

```
  1.  A'length = B'length
  2.  A'length > 0            (i.e. (A'length = B'length and A'length > 0)
  >> [] = [] ++ []
-> [] = [] ++ [].)
```

- For each component, the PRE condition of PBC, the CI, and the initial condition of that task imply the PRE condition of the component. The preconditions of *PRODUCER*, *BUFFER* and *CONSUMER* are all *true*, so there is nothing to prove.

- The POST conditions of the components plus the CI imply the POST condition of PBC.

```
  1.  PRODUCER.to_Q = IN A
  2.  BUFFER.Q = []
  3.  CONSUMER.from_Q = B
  4.  PRODUCER.to_Q = BUFFER.Q ++ CONSUMER.from_Q
  >> B = A
```

4 Task Bodies

In this section, we look at verification of individual tasks. The main points to be covered are

- semantics of entry calls and accept statements and
- preservation of the cooperation invariant.

Let us look at the body of the PRODUCER task first. It will be easy because it does not contain any accept statements. The CONSUMER task would be similar, so we will omit it.

```
task body PRODUCER is
  I: INTEGER := 1;
begin
```

```
    while I <= A'length loop
       --| invariant to_Q = A[1..I-1];
       buffer.write(A(I) --: virtual to_Q;
                   );
       I := I + 1;
    end loop;
    --| to_Q = A[0..A'length];
 end PRODUCER;
```

Aside from the VC connected with the loop, we have the initial VC that the initial value assignment plus the PRE condition imply the precondition of the task body. This is all routine. The only thing to look at is preservation of the cooperation invariant and the precise form of the predicate transformer for the call to `buffer.write`.

The cooperation invariant is required to hold when each synchronization point is reached. Thus, for each entry call/accept rendezvous, the cooperation invariant must be a conjunct of each of the following, in addition to the annotations at those points:

- the precondition of the accept statement (in the accepting task);
- the precondition of the entry call (in the calling task); and
- the postcondition of the entry call (in the calling task).

For example, the precondition of the `write` entry call is as follows.

```
(to_Q = Q) ++ from_Q and (cons(c,to_Q) = cons(c,Q) ++ from_Q )
    and theta
```

where `theta` is the postcondition of the entry call.

The proof that the CI holds after the entry call need only be done once for both the calling and the accepting tasks, because just one piece of code, the entry call, has just been executed for both. The proof is done in the calling process because only it knows whether any of the arguments to the entry call were state variables, and which state variables they were.

To show that the CI holds at the beginning of the rendezvous, it suffices for the following reasons to show that the caller and the acceptor have each individually preserved the CI.

- According to RM 9.11, it is not permitted for both tasks to change the same global variables since their previous synchronization points (and this ban will be enforced by our global annotations).

- Hence, however their code is executed, it is the same as if one task had run first then the other.
- Therefore, it suffices to show that each task separately has preserved the CI since its previous synchronization point.

There may appear to be a problem if an accept statement for an entry that has some `IN OUT` or `OUT` parameters itself contains an entry call or another accept statement: if the semantics of `IN OUT` is call by reference and the corresponding actual parameter is a state variable, then assignment to that parameter between the beginning of the original accept statement and a nested entry call or accept may violate the CI. This is not a real problem, however, because Ada requires and Penelope's aliasing checks guarantee that the behavior of a call be the same whether the semantics is call by reference or call by copy. Hence we can treat `IN OUT` parameter semantics as call by copy. The value of a state variable that is an `IN OUT` or `OUT` variable is not changed until the entry call returns; hence the CI cannot be inadvertently violated in a way that is not detectable inside the accept statement.

Note that although `to_Q` is meant to be an abstraction of `A[1..I-1]`, the abstraction relation need not hold all the time. In fact, the annotations require only that it hold at the top of the loop and at the end of the program. In fact, there need be no formal requirement that any particular abstraction relation ever hold. If the virtual state variables bore no relation to actual variables, then we just could not prove anything about the data transmitted by rendezvous. This looseness is very useful because it permits a clean abstraction without imposing any constraints on the implementation.

(To give fuller argument: Obviously the verifications would be correct if the virtual variables were made actual. But the effect of the program on its original actual variables is the same whether the virtual variables are actual or not. Our use of virtual variables is the same as Owicki's method of auxiliary variables [10].)

The body of the `CONSUMER` task will be as follows. The same concerns apply to the `CONSUMER` task as to `PRODUCER`, so we will make no further comment.

```
task body CONSUMER;
  J: INTEGER := 1;

begin
  while J <= B'length loop
     --| invariant from_Q = B[1..J-1];
     buffer.read(A(J) --: virtual from_Q
                   );
     J := J + 1;
  end loop;
  --| from_Q = B[1..M];
end CONSUMER;
```

Now we go on to the buffer itself.

```
task body BUFFER is
   POOL_SIZE : constant INTEGER := ?; -- any constant > 0;
   POOL      : array(1 .. POOL_SIZE) of CHARACTER;
   COUNT     : INTEGER range 0 .. POOL_SIZE := 0;
   IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
begin
   loop
      --| invariant queue_rep(Q,POOL,COUNT,IN_INDEX,OUT_INDEX);
      select
         when COUNT < POOL_SIZE =>
            accept WRITE(C : in CHARACTER
                    --: virtual stream_in: in out Stream
                            )
            do
               POOL(IN_INDEX) := C;
               --: Q := Q ++ [c];
               --: stream_in := stream_in ++ [c];
            end;
            IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
            COUNT    := COUNT + 1;
      or
         when COUNT > 0 =>
            accept READ(C : out CHARACTER
                    --| virtual stream_in: in out Stream
                             )
            do
               C := POOL(OUT_INDEX);
               --: Q := tl(Q);
               --: stream_out := stream_out ++ [C];
            end;
            OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
            COUNT     := COUNT - 1;
      or
         terminate;
      end select;
   end loop;
end BUFFER;
```

Besides the accept statement, the BUFFER task introduces three other interesting constructs, the select, the when and the terminate.

- `select R1 or ...or Rn` means that the program may execute any of the `R1, ..., Rn`, each of which is either
  - an `accept` statement or
  - an `accept` guarded by a `when`, whose guard has the value true.

  Each `Ri` in the select statement must be of this form. (For `else` alternatives see Appendix A.)

  Of course, an `accept` statement will not be executed unless another task calls it. If a task calls an entry that its owner is not waiting to accept, then the calling task is blocked and the call is put in a queue. These possibilities relate to the possibility of deadlock, discussed in Section 11. They do not affect partial correctness.

  The predicate transformation for a `select` statement is:

$$wp(\texttt{select R1 or ...or Rn;})\phi = wp(\texttt{R1})\phi \wedge \ldots \wedge wp(\texttt{Rn})\phi \qquad (2)$$

- A `when` statement is of the form

```
when b => A;
```

  where `b` is a boolean expression and `A` is an accept statement. It means that this alternative may be taken only if `b` holds. The predicate transformer is

$$wp(\texttt{when b => A;})\phi = \texttt{b} \rightarrow wp(\texttt{A})\phi,$$

  assuming that evaluation of `b` has no side effects.

- The `terminate` alternative means that the task is willing to terminate if the task or subprogram that started it (its *master*) and all other tasks started by that master have terminated or are similarly ready to terminate. The precondition of a `terminate` statement is

$$CI \wedge \left(\bigwedge_i POST_i \rightarrow POST \right)$$

  where $POST$ is the `POST` condition of the task and $POST_i$ are the `POST` conditions of its children.

Now let us look at semantics of accept statements. Consider the declaration and the `accept` statement for `READ`.

```
entry READ(c: OUT character --: virtual stream_out: in out Stream
                );
        --| PRE Q /= [];
        --| OUT (IN Q) = cons(Q,c) ;
        --| OUT stream_out = cons(c,(IN stream_out));
```

```
when COUNT > 0 =>
   accept READ(c : OUT character
           --: virtual stream_out: in out Stream
                              )
   do
      C := POOL(OUT_INDEX);
      --: queue := tl(queue);
      --: stream_out := stream_out ++ [C];
   end;
   OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
   COUNT     := COUNT - 1;
```

The first thing to note is that not all the computation associated with the entry call is contained inside the entry call itself. Intuitively, WRITE takes a character off the queue and returns it. The implementation of the queue is the segment of POOL from OUT_INDEX back to IN_INDEX. The character is not actually taken off the implemented queue until OUT_INDEX is incremented, which is after the end of the accept statement. Before then, it might have been put into B. If Q were just the implemented queue and from_Q were just B[0..J], then this would lead to a violation of the cooperation invariant. It is for this reason that we have introduced the virtual variables Q and from_Q.

Dillon [2] gives as a reason for avoiding virtual variables the fact that using them sometimes requires that an accept statement be given a body when it otherwise would not have one. (For example, it might have no arguments, the mere fact of it being called being sufficient to transmit needed information.) We should invent some convention to avoid creating such unnecessary bodies for accept statements. Either a convention like that above or some sort of virtual body should be used.

The predicate transformation associated with an accept statement is given by:

$$wp(\texttt{accept xxx(...) do S end;})\phi = CI \wedge \forall \ldots (CI \wedge \theta_i \rightarrow wp(\texttt{S})(\theta_o \wedge \phi)) \qquad (3)$$

where θ_i and θ_o are the in and out conditions from the entry declaration and the ellipsis points represent the formal paramenters of the entry.

4.1 Using Package PBC

Package PBC acts like a big task with no task entries and the given PRE and POST conditions. It still has to be treated like a task because it does have globals. We might use PBC inside a procedure like the following one, that occupies its time doing something else while waiting for PBC to do its work.

```
procedure copy_and_power(A: in string, B: out string, n: in integer,
                                two_to_n: out integer)
        --| IN A'length > 0 and A'length = B'length and n >= 0;
        --| OUT B = IN A and two_to_n = 2**n;
  is
  p : integer := 1;
  package PBC is
        ...
  end PBC;
  package body PBC is separate;

 is
  begin
   for i in 1..n loop
      p := p*2;
   end loop
   two_to_2 := p;
   return;
  end copy_and_power;
```

Conceptually, this subprogram consists of two tasks, its own body and PBC. No cooperation invariant is necessary because neither has any task entries. No global variables of PBC are lmods or rmods of the part of copy_and_power that comes after the body of PBC, so copy_and_power needs no globals annotation.

The precondition of the final return statement is that the POST condition of PBC implies the OUT condition of copy_and_power.

```
        (B = in A) -> (B = in A) and (two_to_n = 2**n)
```

If there had been more tasks or a cooperation invariant, this precondition would be that the conjunction of the cooperation invariant and the postconditions of all the proper tasks implies the out condition of the subprogram.

The initial VC of copy_and_power is that the IN condition must imply the conjunction of the precondition of the body and the PRE condition of PBC. The globals condition of PBC is discharged automatically because A and B are not mods of the body of copy_and_power.

From the outside, copy_and_power looks like an ordinary subprogram and nobody else needs to know that it contains any tasking.

5 Separate, Composable Verification of Tasks

Consider the cooperation invariant in the producer-buffer-consumer example.

```
--| CI: PRODUCER.to_Q = BUFFER.Q ++ CONSUMER.from_Q;
```

Suppose that instead of one producer and one consumer there are several of each. For concreteness, say that there are two producers, PRODUCER1 and PRODUCER2, and two consumers, CONSUMER1 and CONSUMER2. Then we would need a different cooperation invariant. For clarity, we might introduce two new state variables in BUFFER, input_stream and output_stream. Modify the entry declarations to read as follows.

```
entry write(c: char);
        --: virtual producer_stream: in out Stream;
        --| OUT input_stream = cons(c, (IN input_stream));
        --| OUT producer_stream = cons(c, (IN producer_stream));

entry read(c: OUT char);
        --: virtual consumer_stream: in out Stream;
        --| OUT output_stream = cons(c, (IN output_stream));
        --| OUT consumer_stream = cons(c, (IN consumer_stream));
```

Then we can write the cooperation invariant as follows.

```
--| CI: shuffle(CONSUMER1.to_Q,CONSUMER2.to_Q,BUFFER.input_stream)
--|      and
--|     shuffle(PRODUCER1.from_Q,PRODUCER2.from_Q,BUFFER.output_stream)
```

The equation

```
input_stream = Q ++ output_stream
```

can be made part of BUFFER's POST condition and loop invariant Q can be left out of the state of BUFFER.

The shuffle predicate is defined as its name implies: l_3 is obtained by interleaving l_1 and l_2.

$$\begin{array}{l} \mathit{shuffle}([],[],[]) = \mathit{true} \\ \mathit{shuffle}(a:l_1, l_2, a:l_3) = \mathit{shuffle}(l_1,l_2,l_3) \\ \mathit{shuffle}(l_1, a:l_2, a:l_3) = \mathit{shuffle}(l_1,l_2,l_3) \\ \mathit{shuffle}(l_1,l_2,l_3) = \mathit{false} \text{ (otherwise)} \end{array} \tag{4}$$

If we again change the number of producers and consumers, we will again have to change the cooperation invariant. This is a problem, because the verifications of each producer and consumer depends on the cooperation invariant, since it appears as an implicit pre- and postcondition in each entry call. We solve this problem by noting that the verification of each individual task depends only on the cooperation invariant being preserved by entry calls. For example, the verification of CONSUMER1 will go through with any cooperation invariant CI that satisfies the following formula.

```
forall c: Char:: CI ->
      CI[CONSUMER1.to_Q <= c:CONSUMER1.to_Q,
                      BUFFER.input_stream <= c:BUFFER.input_stream]
```

Since the variables of CI are not specified, we use the notation for array updates to indicate substitution. Now we have a generic verification of a producer task that we can file away and reuse in a producer-consumer system with any number of producers and consumers.

A minimal CI **constraint** must state the following:

1. the CI is preserved by entry calls;
2. the CI is preserved by execution of code between synchronization points.

Each constraint of this kind can always be expressed in the form

```
forall x1,...,xn :: CI -> CI[x1,...,xn <= f(x1,...,xn)]
```

or

```
forall x1,...,xn,y1,...,yn :: R(x1,...,xn,y1,...,yn) and
         CI -> CI[x1,...,xn <= y1,...,yn]
```

Here R is just the relation between the state at one synchronization point and the next.

6 Declarative Regions as Quasi-Tasks

In the previous sections we have seen that a package that contains tasks may have to be treated as a big task because it contains code that can be executing concurrently with code outside it. The same is true of any declarative region that can have a declarative part. In this section, we summarize what annotations such a declarative region will need in its capacity as a quasi-task and what VCs have to be generated to relate these annotations to the task annotations of its components.

By a *quasi-task*, we mean a declarative region of a program whose denotation will be a state machine. Its state consists of the locations of instruction pointers of all tasks it contains and the values of their state variables. We will go into more detail about exactly what a quasi-task is in Section 15.1. Here, we are interested only in aspects relevant to partial correctness, namely state variables and cooperation invariants.

A quasi-task is what might intuitively be called a process. We use the name quasi-task because it differs technically from the use of the term process in process algebra, as in [5, 9, 12], a process there being a state machine in a particular state, considered from the point of view of how it can be transformed into other processes by participating in communications.

Declarative regions are important in the context of tasking because they may combine component tasks, hide certain task communications, and leave other task communications visible. Accordingly, declarative regions will need two sets of task annotations: external annotations governing externally visible communications and variables; and internal annotations that add supplementary information about the internally visible communications and state variables.

Let us be precise: A *quasi-task* is a

- package,
- subprogram,
- task or
- block.

It will be important to think of a region as a quasi-task if it contains

- task entries visible outside it,
- calls to task entries outside it, or
- uses globals, global variables that are visible outside it and are lmods or rmods of some task inside it.

If it does not do any of these things, it is still a quasi-task, but a trivial one; it does not need any of the annotations peculiar to quasi-tasks and it can just use defaults.

The *components* of a quasi-task are its own body and the quasi-tasks declared in its declarative part including all tasks in the collection associated with any access types for a type that has task components. Semantically, a quasi-task is the parallel composition of its components.

The externally visible annotations of a quasi-task are as follows. Syntactically, these annotations will all be basic declarative items.

- State specification with initial values.

  ```
  --| STATE:
  --|   x:  Sort1 := a; y:  Sort2; ...--| end STATE;
  ```

  For packages, the state does not include the state of components visible in the package declaration, since those states will be visible outside the package. The package state should summarize the state of hidden components for use in the cooperation invariants of containing quasi-tasks.

  State variables do not have to be given an initial value. A state variable that does not have an initial value is considered to have a well-defined but unknown value.

  If there is no state declaration, there are considered to be no state variables.

- Entry and exit conditions. For packages and tasks, these are `PRE` and `POST` conditions. For subprograms, they are the `IN` and `OUT` conditions. Block statements do not have them.

- Global annotations, which are of the form

  ```
  --| GLOBALS A: in, B: out, C: in out;
  ```

  where `A`, `B` and `C` are variables global to the quasi-task. For packages, these declarations need cover only globals of hidden components, not the visible ones.

- Declarations, including declarations of all component quasi-tasks.

- Cooperation invariant constraint:

  ```
  --| CI CONSTRAINT: P;
  ```

  Here, `P` is a generalized formula as described in Section 5. That is, it is like a formula except that it can contain terms

  ```
  CI[x1 <= t1]...[xn <= tn]
  ```

  where `x1`, ..., `xn` are Larch or Larch/Ada variables and `t1`, ..., `tn` are terms. Unlike other annotations, for a package, the CI constraint must subsume the CI constraints of all components.

  Normally, any state variables that are not mentioned in the CI constraint are superfluous and should be eliminated.

  If the CI constraint is absent, the quasi-task simply inherits the CI constraint and the local CI of the quasi-task that contains it.

The annotations should normally occur in this order. The CI constraint comes last because it probably needs to mention state variables of the component quasi-tasks. For packages, all the foregoing annotations must appear in the package declaration.

In addition, each quasi-task may have the following annotations.

- Declarations of all internal components.
- Local cooperation invariant. It may mention only state variables that cannot be changed by any call to an external entry or by a call to any externally visible entry of this quasi-task.

  The cooperation invariant `T.CI` of a quasi-task `T` will be the conjunction of its local and global cooperation invariants (the latter will just be denoted `CI` if `T` has a CI constraint instead of inheriting an explicit global CI).

If the quasi-task is a package, these annotations should occur in the package body.

These are the main annotations relevant to partial correctness. In addition to these annotations, a quasi-task may have exception annotations (as for sequential programs), abort annotations, deadlock annotations, and progress annotations. These other kinds of annotations will be discussed in later sections.

The annotations of a quasi-task `T` generate the following proof obligations.

1. For each component quasi-task `T'` of `T`, if `T'` has a CI constraint, then we must prove the following VC showing that the CI constraint and local cooperation invariant of `T` are compatible with the CI constraint of `T'`.

   ```
   1.  T.CI_CONSTRAINT
   >> T'.CI_CONSTRAINT[CI <= T.CI]
   ```

   For the purpose of this proof, the variables of `CI` can be taken to be just the variables visible in `T`.

2. If `T` is a non-library package, then the following VC is associated with the `POST` condition of `T`.

   ```
   T1.POST and ...Tn.POST -> T.POST
   ```

   Here, `T1`,..., `Tn` are the components of `T`. (This VC is in addition to the postcondition of a package defined in the sequential predicate transformers. `T.POST` is the condition that must hold when the tasks in `T` terminate. The postcondition of the package is the condition that must hold on completion of the execution of the sequence of statements in the package.)

   If the quasi-task `T` is anything but a non-library package, the post condition of the body of `T` is

   ```
   T.CI and (T1.POST and ...Tn.POST -> theta)
   ```

   where `T1`, ..., `Tn` are the components of `T` and `theta` is

- `T.POST` if `T` is a task or a library package;
- the `OUT` condition of `T` if `T` is a subprogram; or
- the postcondition of `T` if `T` is a block.

The reason for distinguishing non-library packages is that only in them is the termination of packages *not* synchronized with the `end` statement of the quasi-task.

3. If `T'` is a component of `T` that is a task, then `T.CI and T'.PRE` is the precondition of the activation of `T`. In particular, the following things hold.
   (a) `T'.PRE` must imply the local CI of `T'`.
   (b) The precondition of the `begin` following the declarative part of `T` is

   ```
   theta and T.CI and T1.PRE and ...and Tn.PRE
   ```

   where `theta` is the precondition of the sequence of statements of `T` (with, for each `i`, `Ti.location` set to `Ti.init`—see Section 8). `T1,..., Tn` are the declared (as opposed to allocated) components of `T`.
   (c) The precondition of a `new` that allocates a task will imply the `CI` of the task doing the allocating and the `PRE` of the task being allocated.
4. Global annotations work as they currently do in Penelope. That is, a task, package, or subprogram must have an appropriate global annotation for any global variable that is an lmod or rmod of the task, package, or subprogram. Component tasks must be considered in computing lmods and rmods.

   The proof obligations related to global annotations are discussed in Section 10.

   Global annotations enforce the mutual exclusion conditions that RM 9.11 imposes on use of global variables not specially designated using pragma `SHARED`.

7 Delay Statements

Delay statements require only that there be a global virtual variable t that increases each time one looks at it and that any delay statement increases by at least the delay amount. The value of t is returned by the function `clock`.

8 Control Flow of Tasks

For a number of purposes, it is necessary to have a language for describing the location of control in a task. In Ada, the location of the control point in a task is defined (and is only well-defined) relative to synchronization points of the task.

Accordingly, we associate with each task an implicit virtual state variable called its *location*. With one exception, the location of a task is the last synchronization point it has passed. The location points of the task are the following.

1. `init`, its location when it is activated.
2. `begin`, the point just after the begin following its declarative part.
3. Each entry call `A`, and its return.
4. Sets of accept statements and terminate alternatives. (These are not synchronization points.)
5. The initiation of each entry call.
6. The end of each `accept` statement.
7. Each `abort` statement.
8. Each reference to a variable declared using pragma `shared`.
9. Allocation of any subtask.
10. Completion of the task.

Quasi-tasks other than tasks have the same locations point, other than `init` (and for non-library packages, `completion`), but do not have an independent location variable. A subprogram is passed the location variable of its caller as an implicit parameter. The location variable of a package or block statement is the location variable of the smallest task or subprogram that contains it.

Duplicate accept statements or entry calls may be distinguished by labels in order to give a more precise description of control flow, but it is not necessary to do so.

The intuitive idea of locations is that a virtual assignment

```
location := α;
```

takes place simultaneously with each synchronization or arrival at a `select` statement (after evaluation of the guards of a selective wait). Some complications arise because of the multiple nature of certain Ada synchronizations, namely task activations, task termination and entries associated with selective waits. Let us examine these one by one.

- Selective wait statements. Consider the statement

  ```
  select when b1 => accept A1 ...or ...or when bn => accept An ...;
  ```

  (Consider an unconditional alternative to be the same as `when true => ....`) After the guards have been evaluated, the task `T` containing the statement has location

  $$\{\texttt{Ai};\ \texttt{bi} = \textit{true}\}.$$

  This is also the code if one of the alternatives is a `terminate` alternative.

- Selective wait with an else.

  ```
  select when b1 => accept A1 ...or ...or
        when bn => accept An ...else ...;
  ```

  Here, T has location

  $$\{\texttt{Ai}; \texttt{bi} = \textit{true} \ \wedge \exists \texttt{T}' \ \texttt{T}'.\texttt{location} = \texttt{W}, \ \texttt{W} \text{ a call to entry } \texttt{Ai}\}.$$

  If this set is empty, then the location of T is not updated. (We say "entry named by Ai" because it may have a label or multiple accepts.)

- Selective wait with a delay. The delay alternative has the same semantics as an else.
- Simple entry calls. The task is at the entry call.
- Simple accept calls. The task is at the corresponding singleton set of accepts.
- Conditional and timed entry calls. One is at the call only if somebody else is at a matching accept.

Using locations, the attributes CALLABLE, TERMINATED, and COUNT can be given straightforward semantics.

8.1 The Theory of Tasks and Locations

In order to talk about blocking and deadlock in a quasi-task, we need a language in which we can talk about all the dependent tasks of a quasi-task and their control points.

The sorts of the theory are the sorts of

- quasi-tasks and
- locations.

A location of a task or subprogram is either a location contained in it or a location of a subprogram that it calls.

The following predicates and relations are provided.

1. The dependency relation on quasi-tasks.
2. The "is a task" predicate.

3. The relation "α is a location of the quasi-task T."

$$\textit{quasi-tasks} \times \textit{tasks}$$

   is provided.

4. A location function from tasks to locations, standing for the current location of the task.

If the body of T is visible and contains entries $\beta_1, \ldots, \beta_m$ and calls subprograms $P_1, \ldots, P_n$, then a transform will be provided to automatically rewrite "α is a location of T" to

$$\alpha \in \{\beta_1, \ldots, \beta_m\} \cup \bigcup_{1 \leq i \leq n} \textit{locations}(P_i).$$

Note that at any time, one can in theory talk about the locations of all dependent tasks of all visible quasi-tasks, even though those dependent tasks are not visible. To actually do such a thing is usually not necessary, and a program written in such a way that it is necessary to do so is poorly designed.

9 Exceptions and the Abort Statement

It appears that to a large extent, exceptions can be handled as in sequential Ada. In particular, handled exceptions can be treated as for sequential programs.

The main difference between exceptions in tasks and exceptions in other declarative regions is that an unhandled exception in a task is not propagated, but simply causes that task to become completed.

A task may contain a promise annotation for unhandled exceptions (they need not be distinguished since they are not propagated).

```
--| ON UNHANDLED EXCEPTION PROMISE theta;
```

The condition `theta` is the precondition of any unhandled exception. It must imply the postcondition of the sequence of statements of the task containing it.

Executing an **abort** statement is somewhat analogous to raising an unhandled exception in the task aborted. Aborting a task leaves that task in an "abnormal state." The task will become completed by the time it reaches its next synchronization point. The fact that we do not exactly know when the aborted task will be completed is reflected in the semantics by our not knowing the values of any global or state variables that can be modified in the synchronization region of the given task. The semantics of the abort statement is as follows.

1. Each task may have an optional abort annotation. It may not mention any global variables. It must imply the postcondition of the sequence of statements of the task. It must hold at each synchronization point of the task.
2. The precondition of a statement `abort T` in a task `S` is the conjunction of its postcondition and `S.CI` with all global `OUT` variables of `T` universally quantified.

An abort annotation may contain non-global state variables of the aborted task, because other tasks may regard the values of these variables as unchanged since the aborted task last reached a synchronization point.

If a task `S` calls an entry of a task `T` that is already completed or becomes completed during the corresponding `accept` statement, the exception `TASKING_ERROR` will be raised in `S`. In establishing the precondition of this exception in `S`, the completion condition of `T` (postcondition of the sequence of statements of `T`) may be assumed. All of this will be a conjunct of the precondition of the call to the entry of `T`.

We expect that normally there will be no `abort` statements and `TASKING_ERROR` will never be raised by a synchronization with a completed or abnormal task. In this case, there must be no `abort` or `UNHANDLED EXCEPTION` annotations and their conditions are considered to be `false`. In this case, rather than requiring to prove as part of the precondition of each entry call that the task being called has not terminated, we will require the proof of deadlock freedom to show that no entry of a completed task is ever called.

In our semantics, it does not seem useful to distinguish *abnormal* from *completed*.

We remark that since an unhandled exception in a task will not be propagated, but will cause just the task it occurs in to terminate, rather than being handled later or causing the whole program to abort, it can lead to a program that terminates with an unexpected result. This possibility makes the policy of ignoring certain predefined exceptions much less comfortable than in purely sequential programs, because it can make a program return bad results without any warning, rather than just aborting it. It would seem that the thing to do would be to require a proof that no such exception can arise in a task without being handled.

10 Shared Variables

Ada RM 9.11(4,5) requires that tasks observe a form of mutual exclusion in using global variables other than those designated using pragma `SHARED`. That is, a program must be written so that a non-`SHARED` global variable written by any task may not be read by another task while control in the writing task is in between the same two synchronization points as the write to the variable. A read of a global variable similarly excludes any other task from writing it.

We propose the following notation system for enforcing this mutual exclusion.

Each task, package or subprogram that uses a global variable `V` must have a corresponding global annotation, just as subprograms do now.

```
U : IN;
V : OUT;
W : IN OUT;
```

If `V` is an lmod (resp. rmod) of a task `T`, not counting component tasks, `T` has a virtual variable `T.V'read` (resp. `T.V'write`). The default initial values of `T.V'read` and `T.V'write` are both true when the variables exist. (Hence, in the case where only one task ever uses a given global variable, no annotations need ever mention `T.V'read` or `T.V'write`.)

A virtual assignment to `T.V'read` or `T.V'write` may be made immediately after any synchronization point, but nowhere else. For each pair of components `T1` and `T2` of `T` that have tasking global annotations for `V`, each of the following that is applicable is an implicit conjunct of the `CI` of `T`.

```
not(T1.V'read and T2.V'write)
not(T1.V'write and T2.V'read)
not(T1.V'write and T2.V'write)
T1.V'read implies T.V'read
T1.V'write implies T.V'write
T2.V'read implies T.V'read
T2.V'write implies T.V'write
```

`T.V'read` (resp. `T.V'write`) is a precondition of any occurrence of `V` as an lmod (including in an annotation; resp. rmod).

11 Freedom from Deadlock

Given our language for talking about locations, it is easy to define what it is for a quasi-task to be blocked. To prove freedom from deadlock, one need only show that the quasi-task in question (probably one with no external entry calls or externally visible entries) is never blocked. That is, one just needs to show that the statement "`T` is not blocked" is a consequence of the cooperation invariant of `T`. (Typically, this non-blocking statement would be a conjunct.)

Here is the definition of blocking. Note that it does not need the whole language of locations, only the idea of being at an entry call, at a select statement (set of open alternatives) or at or after completion (= at termination).

1. Two locations *match* if one is an entry call, one is an acceptance set, and the accept

corresponding to the call belongs to the set.

$$match(l_1, l_2) = is\_entry(l_1) \wedge is\_accept\_set(l_2) \wedge entry\_of(l_1) \in entries(l_2)$$

2. A quasi-task is terminable if either it has already terminated or it and all its dependents are completed, terminated, waiting on terminate alternatives, or never initiated (for unallocated members of the collection associated with an access for a type with task components).

$$\begin{aligned} & terminable(T) = \\ & \quad (location(T) = T.terminated) \vee \\ & \quad ((location(T) = T.completed \vee \\ & \qquad terminate \in accepts(location(T)) \vee \\ & \qquad location(T) = T.init) \wedge \\ & \qquad \forall S \in components(T)\ (terminable(S))) \end{aligned}$$

A quasi-task T is terminable if it and its dependents are not preventing its master from terminating. Besides being at a terminate alternative, it (and its dependent tasks) may also be at some accept statements, so termination may not be the only thing it can do.

3. A task is *blocked* if it is not terminable and none of the locations of itself and its dependent tasks match.

$$\begin{aligned} & blocked(T) = \\ & \quad \neg terminable(T) \wedge \\ & \quad \forall S, S' \in dependents(T)\ (\neg(match(T.location, S.location) \wedge \\ & \qquad \neg(match(S.location, S'.location)) \end{aligned}$$

Note that the definition of blocking involves talk about what arbitrarily deeply nested tasks are doing. Normally, in a well-constructed program, conditions under which T may be blocked can be described solely in terms of locations of its visible tasks. In a proof of deadlock freedom, the CI of each quasi-task will contain a conjunct

$$blocked(T) \leftrightarrow \ldots$$

or

$$blocked(T) \rightarrow \ldots$$

characterizing or giving a necessary condition for blocking in terms of locations of itself and of components visible outside it.

To prove deadlock freedom of the producer-buffer-consumer package, PBC, we just need to make the following observations.

```
blocked(PRODUCER) = (PRODUCER.location = BUFFER.write)


blocked(CONSUMER) = (CONSUMER.location = BUFFER.read)
blocked(BUFFER) = (BUFFER.write in BUFFER.location or
    BUFFER.read in BUFFER.location)
```

Deadlock freedom, `not blocked(PBC)`, follows immediately from these. The actual proof would be just to write this as the cooperation invariant of `PBC`. No other annotations are needed for the proof of deadlock freedom. (The loop invariants can be taken to be just `true`.)

12 Task Formal Parameters

Because tasks are syntactically objects in Ada, they may be passed as parameters to subprograms, though only as `IN` parameters. Even though a task is an `IN` parameter, however, its state can change during the call because of rendezvous it may have with other tasks both inside and outside the subprogram.

A subprogram with a task parameter is, while it is executing, very much like a package whose declaration contains a task body declaration. The main difference is that the task need not be in its initial state when the subprogram is called, but that is not a major change.

`IN`, `OUT`, and `RETURN` annotations of subprograms with task parameters (or parameters with task components) will naturally be stated in terms of the locations and state variables of the parameter tasks and their dependent tasks. In addition, the subprogram will, like a package, have a `CI` constraint. When the subprogram is called, the `CI` of the calling quasi-task must satisfy the subprogram's `CI` constraint, modulo parameter substitutions.

Global variables, deadlock, and liveness considerations are likewise analogous to those for packages that contain externally visible tasks.

On the whole, the situation here is simpler than for generic procedure subprogram parameters, because fixing the type of a task fixes just what its body is, including any use of global variables, not just the types of the parameters to its entries.

13 Verification without Cooperation Invariants

The use of location variables supports a verification methodology that can avoid or minimize use of virtual variables and explicit cooperation invariants. In this section we will present a brief sketch of this methodology, which is based on that of [14] and is similar to [1].

The rules for this methodology are as follows.

1. Annotations in a given task may mention state variables of any visible quasi-task.
2. The changes that each task makes in its own state variables must not invalidate any assertions that other tasks make about them.

According to our model of concurrent Ada computation in Section 15.1, at any time, just one task will be running, and the rest will all be waiting at synchronization points. Consequently, to be consistent with the model, each task need only preserve validity of the preconditions of synchronization points of other tasks. These conditions are the PRE and POST conditions of entries and tasks, IN and OUT conditions of entries, pre- and post- conditions of the sequence of statements of quasi-tasks (only the precondition for non-library packages).

Each such condition requires an invariance check. The simplest way to do that with the machinery we all ready have is to implicitly add the following conjunct to the CI for each such condition that mentions state variables of tasks other than the one to which it belongs.

```
T.location = L -> P
```

Here L is a synchronization point of T, and P is the precondition of that synchronization point.

Of course, we can achieve the same logical effect by simply adding these conjuncts to the CI ourselves. The advantage of putting them in the code instead of the CI is to make obvious the relation of the assertions to the code that establishes them. It also reduces the need to explicitly mention locations in annotations.

As an example, the following is PBC annotated in the fashion described here. We use circle(POOL,I,J) to denote the list of elements of POOL as a circular queue, starting at I and going to J-1, around the corner if J $\leq$ I.

```
task BUFFER is
    --| POST: A[1..P.I] = B[1..C.J] ++
    --|             circle(POOL,OUT_INDEX,IN_INDEX);

   entry READ (C : out CHARACTER);
    --| IN A[1 .. P.I] = B[1 .. C.J] ++
    --|         circle(POOL,OUT_INDEX,OUT_INDEX) ++ [C];
    --| OUT no change of variables;
    --| OUT A[1 .. P.I] = B[1 .. C.J] ++
    --|             circle(POOL,OUT_INDEX,(IN_INDEX+1 mod POOL.len);
   entry WRITE(C : in  CHARACTER);
    --| IN A[1 .. P.I] = B[1 .. C.J] ++
    --|         circle(POOL,OUT_INDEX,OUT_INDEX) ++ [C];
    --| OUT no change of variables;
    --| OUT A[1 .. P.I] = B[1 .. C.J+1] ++
    --|             circle(POOL,(OUT_INDEX+1 mod POOL.len),IN_INDEX);
end;

task body BUFFER is
```

```
      POOL_SIZE : constant INTEGER := 1;
      POOL      : array(1 .. POOL_SIZE) of CHARACTER;
      COUNT     : INTEGER range 0 .. POOL_SIZE := 0;
      IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
   begin
      --| P2
      loop
         --| invariant A[1..P.I] = B[1..C.J] ++
         --|                  circle(POOL,OUT_INDEX,IN_INDEX);
         select
            when COUNT < POOL_SIZE =>
               accept WRITE(C : in CHARACTER) do
                  POOL(IN_INDEX) := C;
               end;
               IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
               COUNT    := COUNT + 1;
         or
            when COUNT > 0 =>
               accept READ(C : out CHARACTER) do
                  C := POOL(OUT_INDEX);
               end;
               OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
               COUNT     := COUNT - 1;
         or
            terminate;
         end select;
      end loop;
   end BUFFER;
```

14 Control vs. Data in Verification of Concurrent Programs

Intuitively, what a concurrent program does can be divided into two parts: how its control points (locations) move around as the program is executed, and how the values of the program variables (data) change.

The proof of data correctness corresponds roughly to partial correctness, the proof that embedded assertions about the values of data variables hold whenever they are encountered. The most important parts of the control flow verification are deadlock freedom (the proof that it is always possible for some task to be running) and progress properties (the proof that certain control points will eventually be reached). If the program actually terminates, partial correctness, deadlock freedom, and progress constitute a proof of total correctness.

The distinction between control and data verification is not a hard and fast one. Deadlock freedom, for example, is also proved using partial correctness methods, but the assertions used talk mainly about the location of control points and only incidentally about values of data variables. Even the use of control locations could be paraphrased, though with some loss of clarity. Nevertheless the distinction between control and data is a useful and important one.

For example, the verification of the systolic matrix multiplication algorithm also divides neatly along the control-data line. The partial correctness proof shows that, as long as the system of entry calls can be made, the product of the two input matrices will be computed. That proof refers only to the data variables. The proof of deadlock freedom refers only to control variables, and not to the data variables. The liveness (termination) proof is trivial.

In general, the distinction between control flow and data manipulation is useful for the following reasons. First, the effect on data can probably be defined as a fairly classical kind of mathematical function or relation, whereas control flow analysis tends to be a less classical, more combinatorial problem. Second, the effect of a program on data is more tolerant of underspecification. Except in a very robust program, if control flow is underspecified, it will turn out to be impossible to prove either deadlock freedom or progress properties.

This distinction between data-oriented and control-oriented aspects of verification corresponds to the distinction between control path and data path in hardware design and verification. We consider the ease with which our method supports separate specification and verification of control and data to be one of its strong points.

15 Proving General Properties of Quasi-Tasks

It is often desirable to verify some properties of a system that cannot be expressed directly by Larch/Ada annotations. For example, in a message passing system, we would like to say that every message received will eventually be passed on.

A common language for expressing such properties is temporal logic. It extends first order logic by adding an operator *unless*:

$\phi \; unless \; \psi = \phi$ will be true from this moment until ψ becomes true (if it ever does).

From this operator can be derived two other operators, $\Box$ "from now on," and $\Diamond$ "eventually."

$$\Box\phi = \phi \; unless \; false$$

$$\Diamond\phi = \neg\Box\neg\phi$$

We can express our message passing property as follows.

$$\Box\forall m \;\; (receive(m) \rightarrow \Diamond resend(m))$$

In this section, we will outline what it means for a statement of temporal logic to be true of a quasi-task, and how one would go about proving such statements.

15.1 A State Machine Model for Ada Tasking

Before we say how to prove temporal properties of Ada tasking programs, we should first say what they mean. That is, we will associate with an Ada quasi-task a model for temporal logic, and say that a temporal logic formula is true of the quasi-task iff it is true in the associated model.

A natural, and the most commonly used, model for temporal logic is the *state machine*. For our present purposes, a state machine is given by a collection of state variables, taking values in suitable sorts, and a transition relation, $R(s, s')$ relating the current state (assignment of values to state variables) s to states s' that are accessible from it. A *run* of a state machine is a finite or infinite sequence $s_0, s_1, \ldots$ of states such that for each $i \geq 0$, $R(s_i, s_{i+1})$.

If s is a state, we say that a formula ϕ of first order logic is true in s, $s \models \phi$, iff ϕ is true (in the many-sorted model for the sorts of the state variables) when the state variables are replaced by their s-values. We define as follows the property of a temporal logic formula being satisfied by a run. If ϕ is a first order formula, then

$$s_0, s_1, \ldots \models \phi \iff s_0 \models \phi.$$

If ϕ and ψ are temporal logic formulas, then

$$s_0, s_1, \ldots \models \phi \mathit{\ unless\ } \psi$$

iff

$$s_0, s_1, \ldots \models \psi$$

or

$$s_0, s_1, \ldots \models \phi$$

and

$$s_1, s_2, \ldots \models \phi \mathit{\ unless\ } \psi.$$

Satisfaction is defined for quantifiers and connectives in the following way.

A state machine satisfies a temporal logic formula ϕ iff every run satisfies the formula.

With an Ada quasi-task `T`, we associate the following state machine.

- Its state variables are all variables visible from inside the quasi-task and the location variables of the quasi-task and its dependent tasks.
- The transition relation relates pairs of states s and s' such that
  1. except for exactly one location variable, s and s' give the same values of location variables; and
  2. a program state in which the program state variables have the values given by s' can be reached from one in which they have the values given by s by executing only steps of the task `S`, whose location changes from that of s to that of s', in such a way that no control point of `S` is passed along the way.

In terms of the actual Ada program, it means that for the purpose of temporal modeling, the location of a task is the last control point it passed, and the values of its variables are the values they had when it passed that point.

According to this point of view, we do not take account of any changes of program variables between control points. For example,

$$\mathtt{T} \models \Box\,(\mathtt{x} = 0)$$

may be true even though **x** is temporarily set to 1 between control points. This point of view seems appropriate because such temporary changes have no effect on the externally observable behavior of the program.

This state machine model is equivalent to a process logic model in which each synchronization event has attached to it the location and values of the state variables of all the tasks involved.

We plan eventually to prove that the methodology for verifying concurrent Ada programs given in this paper is sound relative to this state machine model.

15.2 Verifying Temporal Assertions[1]

We do not propose that Penelope annotations be written in temporal logic. Rather, we propose to write

$$\mathtt{T} \models \phi \tag{5}$$

to indicate that a quasi-task T satisfies a temporal logic formula ϕ. We will outline rules for deriving such a temporal logic assertion about T from a partial correctness verification of T and temporal assertions about the components of T. Furthermore we will give special rules for deriving assertions of the form (5) only when ϕ is of one of the following special forms: simple safety assertions

$$\psi \rightarrow \Box\theta$$

and progress assertions

$$\psi \bigwedge_i \mathtt{Ti}.\mathit{location} = \alpha_i \rightarrow \Diamond \bigvee_{\beta \in B} \bigwedge_i \mathtt{Ti}.\mathit{location} = \beta_i,$$

where Ti range over tasks whose locations are visible from T, and α_i and β_i are locations of Ti. The thesis is that the temporal assertions that are useful in the actual development of programs are of these kinds and that all other interesting temporal assertions about a program can be derived from temporal assertions of these kinds, possibly with the assistance of history variables. We do not know whether there is a theorem to back up this thesis.

[1]The approach taken in this section derives from a consultation with F. Schneider.

15.2.1 Simple Safety Properties

These properties are easy to express and prove.

$$T \models \phi \rightarrow \Box\psi \tag{6}$$

follows if ϕ implies `T.PRE` and `T.CI` implies ψ.

15.2.2 Progress Properties

First consider a quasi-task `T` that has no dependent tasks and that calls no subprograms (it may call entries of or accept calls from any task or subprogram). Proving that `T` will eventually reach a location in a given set, provided all entry calls satisfy their preconditions and terminate, is similar to proving termination in sequential programs: Annotate each loop in `T` that decreases with each execution of the loop, and can only increase either outside the loop or else when a location in the given set is reached. We can use this idea to prove reachability without passing through certain rendezvous by giving those rendezvous a `false` as a precondition.

Suppose we have proved that `T` satisfies a progress property such as

$$\mathtt{T} \models \phi \rightarrow \Diamond\mathtt{T.location} \in A. \tag{7}$$

If we want to show that

$$\mathtt{S} \models \phi \rightarrow \Diamond\mathtt{T.location} \in A$$

where `S` is a quasi-task that has `T` as a component satisfies, then we must show that

1. other components of `S` leave ϕ invariant;
2. `S` does not deadlock; and
3. from any configuration of locations, `S` will reach a configuration in which each dependent task is blocked or waiting for a rendezvous with `T`. This statement can be proved by induction on the number of components of `S`, giving rendezvous with `T` a precondition of `false`. (This condition can be weakened, but some form of it is necessary because Ada does not require weakly fair scheduling.)

Applying this method inductively, we can prove arbitrary progress assertions. We can also prove more general liveness assertions of the form

$$\mathtt{T} \models \phi \rightarrow \Diamond\left((\mathtt{T.location} \in A) \wedge \psi\right),$$

because this follows from the pure progress assertion (7) and the simple safety property

$$\mathtt{T} \models \phi \rightarrow \Box\left((\mathtt{T.location} \in A) \rightarrow \psi\right).$$

As an example, again consider the `PBC` package.

```
task body PRODUCER is
  I: INTEGER := 1;
begin
  while I <= A'length loop
     --| invariant true;
     --| termination A'length - I;
     buffer.write(A(I));
     I := I + 1;
  end loop;
end PRODUCER;

task body CONSUMER;
  J: INTEGER := 1;
begin
  while J <= B'length loop
     --| invariant true;
     --| termination B'length - J;
     buffer.read(A(J));
     J := J + 1;
  end loop;
end CONSUMER;

 task body BUFFER is
    POOL_SIZE : constant INTEGER := ?; -- any constant > 0;
    POOL      : array(1 .. POOL_SIZE) of CHARACTER;
    COUNT     : INTEGER range 0 .. POOL_SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
 begin
    loop
       --| invariant true;
       select
          when COUNT < POOL_SIZE =>
             accept WRITE(C : in CHARACTER)
             do
                POOL(IN_INDEX) := C;
             end;
             IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
             COUNT    := COUNT + 1;
       or
          when COUNT > 0 =>
             accept READ(C : out CHARACTER)
             do
                C := POOL(OUT_INDEX);
             end;
             OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
```

```
                COUNT     := COUNT - 1;
          or
             terminate;
          end select;
       end loop;
    end BUFFER;
```

Liveness properties for the buffer are the most interesting. Because

```
0 <= COUNT <= POOL_SIZE,
```

we can use COUNT and POOL_SIZE - COUNT as counters to show that either a READ, a WRITE or termination will eventually take place, assuming cooperation by the environment. It is the possible to show that in any environment that permits BUFFER to run occasionally, everything BUFFER reads will eventually be written, and BUFFER will always eventually read another character, unless it terminates first.

Now, we can also show that each of PRODUCER and CONSUMER eventually reaches completion provided they do not permanently block. On the other hand, if no environment task (i.e., any task other than PRODUCER or CONSUMER) ever calls the entries of BUFFER, the latter will always eventually block waiting for such a call or for the completion of PRODUCER and CONSUMER. Taken as a whole, PBC does not deadlock, hence in it each of PRODUCER and CONSUMER eventually completes. At that point BUFFER cannot receive any more read or write calls, so its progress property implies that it must complete. Hence the quasi-task PBC completes.

15.3 An Overall Development Method for Concurrent Programs

We would envisage developing a verified concurrent program in the following way.

1. Start with a system specification in terms of temporal logic.
2. Use a design development/verification system to develop a system design giving a structure in terms of the main quasi-tasks, control points, invariants, and progress conditions that suffice to imply the overall specification.
3. Use Penelope to implement this design in code and verify that implementation.

16 Bibliography

[1] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM TOPLAS*, 2(3):359–385, 1980.

[2] L. K. Dillon. An isolation approach to symbolic execution-based verification of Ada tasking programs. *Journal of Systems and Software.*

[3] R. Gerth and W. P. de Roever. A proof system for concurrent Ada programs. In *Science of Computer Programming 4*, pages 159–204, Amsterdam, 1984. Elsevier Science Publishers B. V. (North Holland).

[4] Matthew Hennessy. Proving systolic systems correct. *ACM TOPLAS*, 8(3):344–387, 1986.

[5] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, Englewood Cliffs, NJ, 1985.

[6] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *IFIP '83*, pages 321–332, Amsterdam, 1983. North Holland.

[7] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[8] L. Lamport. The temporal logic of actions. Technical report, DEC SRC, 130 Lytton Avenue, Palo Alto CA 94301, December 1991.

[9] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1980.

[10] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of ACM*, 19(5):279–284, 1976.

[11] W. Polak. Predicate transformer semantics for Ada, release 1.6. Technical report, ORA, October 1992. ORA internal report.

[12] W. Polak. Semantics and verification of Ada tasking programs. Technical report, ORA, April 1993. ORA internal report.

[13] W. Polak. Verification examples of tasking programs. Technical report, ORA, February 1993. ORA internal report.

[14] R. D. Schlichting and F. B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM TOPLAS*, 6(3):402–431, 1984.

A Semantics of Ada Tasking Primitives

Task termination

$$wp(\texttt{terminate}) = CI \wedge \left(\bigwedge_i POST_i \rightarrow POST\right)$$

where $POST$ is the POST condition of the given task, and $POST_i$ are the POST conditions of its child quasi-tasks.

Entry call

$$\begin{array}{l} wp(\texttt{T.enter(...)})\phi = \\ \quad CI \wedge \theta_i' \wedge \\ \quad\quad \forall \ldots (INCI \wedge IN\theta_{pre} \wedge \theta_o' \rightarrow \phi \wedge CI) \end{array}$$

where θ_i' and θ_o' are the in and out conditions of the entry declaration and θ_{pre} is its PRE condition, with the appropriate substitutions for formal parameters.

Accept statement

$$wp(\texttt{accept enter(...) do S end enter;})\phi = \forall \ldots \ (\theta_{pre} \rightarrow wp(\texttt{S})(\theta_o \rightarrow \phi))$$

where the ellipsis points represent the formals.

Delay statement

$$wp(\texttt{delay E;})\phi(time) = \forall s \geq time + E\ \phi(s)$$

Selective wait

$$wp(\texttt{select S1 or ...or Sn;})\phi = wp(\texttt{S1})\phi \wedge \ldots \wedge wp(\texttt{Sn})\phi$$

$$wp(\texttt{select S1 or ...or Sn else S;})\phi = wp(\texttt{S1})\phi \wedge \ldots \wedge wp(\texttt{Sn})\phi \wedge wp(\texttt{S})\phi$$

Select alternative

$$wp(\texttt{when b => S})\phi = \texttt{b} \rightarrow wp(\texttt{S})\phi$$

Conditional entry call and timed entry call

Like selective wait with else.

Priorities

Adjoin to the precondition of any statement in a task a condition guaranteeing that all tasks of higher priority are blocked. These conditions would be similar to conditions in deadlock annotations, but they must guarantee that tasks are blocked rather than that they are unblocked. This extra part of the precondition need only be adjoined at synchronization points.

Task and entry attributes

`T'CALLABLE` and `T'TERMINATED` probably require annotations in the task giving conditions that must hold for the task to be callable, terminated or abnormal. Then, say if `T'CALLABLE` is true then the callable condition must hold, and if it is false, then either the terminated or abnormal condition must hold.

`E'COUNT` can be modeled using the waiting-on-an-entry-call conditions used in deadlock analysis. If `E'COUNT` is n, then the wait conditions for entry `E` must be true for n processes.

Abort statement

As currently envisaged **abort** will just set a global flag that the relevant task has been completed. The cooperation invariant must be checked.

$$wp(\texttt{abort T})\phi = (\texttt{CI} \wedge \phi)[T'completed <= true]$$

B Concurrent Sieve of Eratosthenes

Given the set S of integers from 2 to N, represented by a bitstring, we let a collection of tasks

```
        sieve(i), 1 <= i <= sqrt(N),
```

(where `sqrt(N)` is the integer part of the square root of N) each delete all proper multiples of i from the set S. In the end, S is the set of primes not greater than N.

The following is intended to be a continuous piece of code—the comment in the middle about CI constraints is not meant to be a real break. The order of the code is not important except that the declarations must precede the CI.

```
N: integer;
type Set is array(2..N) of Boolean;
S : Set := (others => true);

package Sieve_Pak is
   --| global S in out true;
   --| PRE: S = 2..N;
   --| POST: S = {n in 2..N | n prime};
end Sieve_Pak;

package body Sieve_Pak is

task S_handler
  --| global S in out true;
is
  entry delete(n: in integer --: virtual param remd;
                );
        --| OUT S = IN S -- {n};
        --| OUT remd = IN remd ++ {n};
end S_handler;

task type sieve_type
  --| STATE n:Int, init: Bool, removed: Int -> Bool;
  --| POST forall k >1 :: k*n <= N -> removed(n*k);
  --| CI constraint:
  --|   forall k >1 :: k*n <= N <->
  --|      (CI -> CI[S <= S --{k*n}, removed <= removed ++ {k*n}])
is
  entry get_n(m: integer);
```

```
        --| OUT n = m and init;
end sieve_type;
```

An individual sieve task needs to know only that the CI is preserved by removing a multiple of its number n from S.

```
type sieve_array is array(2..sqrt(N)) of sieve_type;
sieve : sieve_array;

--| CI: forall k in 2..sqrt(N) :: (sieve(k).init -> sieve(k).n = k);
--| CI: forall k in 2..N ::
--|          (S[k] = forall n in 2..sqrt(N):: not removed(n)[k]);

task body sieve_type is
  n: integer;
  begin
   accept get_n(n) do n := n; --: init := true;
        end get_n;
   for k in 2..N/n loop
        --| invariant forall k' in 2..(k-1) :: not S(n*k)
        S_handler.delete(k*n --: virtual arg removed;
                                   );
   end loop;
end sieve_type;

task body S_handler is
 begin
   loop
        select
           accept delete(n --: virtual arg remd
                                   )
                do S[n]:= false; --: remd[n] := true;
                end;
        or
           terminate
        end select;
   end loop;
end S_handler;

begin -- Sieve_Pak
  for m in 2..sqrt(N) loop
        sieve(m).get_name(m);
  end loop;
end Sieve_Pak;
```

C Systolic Matrix Multiplication

We present an example verification of a systolic matrix multiplication algorithm. This algorithm is modified from that given in [13, 12] (derived from [4]) so that it can support multiplication of a continuous stream of pairs of matrices. As it is here, we have only annotated it for one multiplication—we plan to upgrade later. For now, we hope that the annotations make it clear that the effect of this system of tasks really is to multiply two matrices. The process logic annotations do nothing of the kind; in fact, they tell the reader no more than the code itself does.

This system finds the product $c = a * b$ where a is an $m \times n$ matrix and b is an $n \times p$ matrix. The main work is done by an $m \times n \times p$ array of cells $\mathtt{mtx}(i, j, k)$. Each cell $\mathtt{mtx}(i, j, k)$, $i, j, k \neq 1$, does the following.

- Fetch the value $a_{i,j}$ from $\mathtt{x}(i, j, k-1)$.
- Fetch the value $b_{j,k}$ from $\mathtt{mtx}(i-1, j, k)$.
- Fetch the partial sum for $c_{i,k}$,

  $$\sum_{j'=1}^{j-1} a_{ij'} b_{j'k}$$

  from `mtx (i, j - 1, k)` and add to it the product $a_{ij} b_{jk}$.

If $k = 1$ (resp. $i = 1$), then $\mathtt{mtx}(i, j, k)$ receives a_{ij} (resp. b_{jk}) from a separate input task. If $j = 1$, then the initial partial sum is just 0. Figure 1 illustrates the function of a cell.

We will first present a slightly simplified version of the system that just multiplies one pair of input matrices and returns the result. This example shows just how simple it is to annotate the system in a way that shows that in fact it does multiply two matrices. In the following subsection, we will make small changes to the program, but considerable additions to the annotations, so that the new program is shown to multiply pairs of matrices in two input streams, with later inputs allowed to start before earlier outputs have been produced.

```
m: constant integer = ...;
n: constant integer = ...;
p: constant integer = ...;

type ma is array(1..m,1..n) of integer;
type mb is array(1..n,1..p) of integer;
```

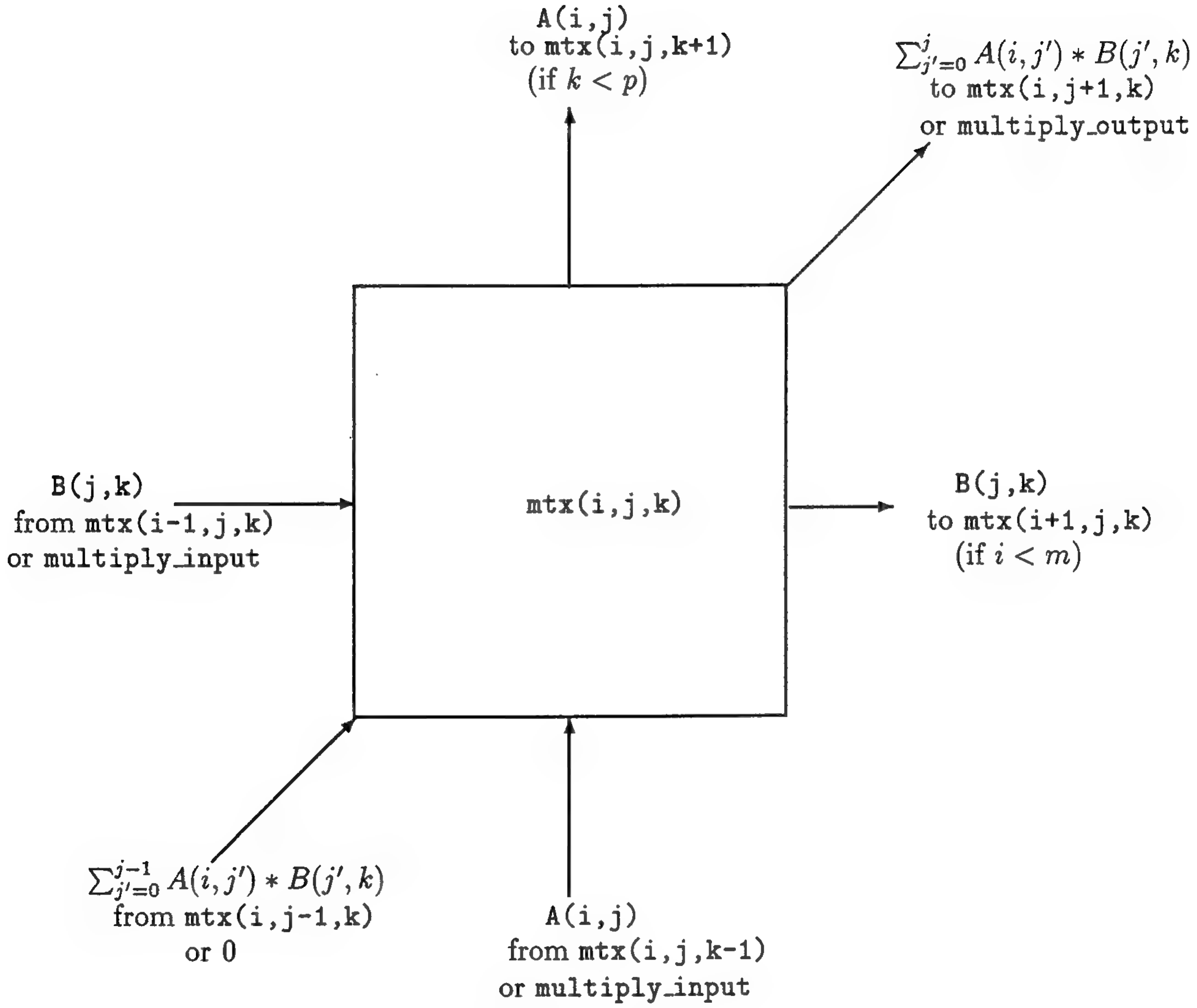

Figure 1: A Cell

```
type mc is array(1..m,1..p) of integer;

        ...

package matrix is

task multiply_input
  --| STATE: A:ma, B:mb;
is
  entry A_in(x: in ma);
  --| OUT A = x;
  entry B_in(y: in mb);
  --| OUT B = y;
end multiply_input;


task multiply_result is
  entry result(z: out mc);
  --| OUT z = A*B;
end multiply_result;
```

Here is the package body. It starts with the declarations of the cell tasks that do all the work. Then we can give the CI.

We define

$$partial\_mult(A,B,i,j,k) = \sum_{j'=1}^{j} A_{i,j'} * B_{j',k}$$

```
package body matrix is

A: ma; B:mb;

task type cell is
 --| STATE: a, b, c, ii, jj, kk : Int;

    entry place(x,y,z:in integer);
        --| OUT (ii,jj,kk) = (x,y,z);
    entry a_in(x: in integer);
        --| IN x = A[ii,jj];
    entry b_in(y: in integer);
        --| IN y = B[jj,kk];
    entry c_in(z: in integer)
        --| IN z = partial_mult(A,B,ii,jj,kk);
```

```
end cell;

type multiplier is array(1..m,1..n,1..p) of cell;

mtx: multiplier;
--| CI: forall i in 1..m, j in 1..n, k in 1..p ::
                mtx(i,j,k).after(place) -> mtx.(ii,jj,kk) = (i,j,k);

task body multiply_input is separate;
task body distribute_input is separate;
task body multiply_result is separate;
task body cell is separate;

begin
  for i in 1..m loop for j in 1..n loop for k in 1..p loop
        mtx(i,j,k).place(i,j,k);
  end loop;
  --| forall (i,j,k) in (1..m,1..n,1..p) :: mtx(i,j,k).(ii,jj,kk) = (i,j,k);
end matrix;
```

Next we present the bodies of the components.

```
task body multiply_input

is
  A: ma; B: mb;
begin
    select
      accept A_in(x) do A := x; end A_in;
    or
      terminate;
    end select;
    accept B_in(y) do B := y; end B_in;
    for i in 1..m loop for j in 1..n loop
            mtx(i,j,1).a_in(A(i,j));
    end loop; end loop;
    for j in 1..n loop for k in 1..p loop
        b_in(B(j,k));
    end loop; end loop;
end multiply_input;
```

Note that in `multiply_input`, no annotations are really needed for partial correctness. The important thing to know is just that all the rendezvous in it do take place, which is deadlock freedom and progress.

The task `multiply_result` gets the product from the array of cells `mtx(i,n,k)`, `1 <= i <= m`, `1 <= k <= p` and makes it available for an external call to fetch.

```
task body multiply_result is
  C: mc;
  begin

    for i in 1 .. m loop for k in 1 .. p loop
        --| invariant forall i',j'::
                (1,1) lex (i',k') lex (i,k-1) -> C(i',k') = (A*B)(i',k');
        mtx(i, n, k).sum(C(i,k));
    end loop; end loop;
    accept result(z) do z := C; end result;
  end multiply_result;
```

Next we presnt the cells.

```
task body cell is
    aa, bb, cc : integer;
    ii, jj, kk : integer;
    have_a, have_b : boolean;
begin
    accept place (x, y, z : integer)
        do
        ii := x; jj := y; kk := z;
        --| initialized := true;
    end;

        select
          accept a_in(x) do aa := x end;
        or
          terminate;
        end select;
        accept b_in(x) do bb := x end;
        if kk < p then mtx(ii,jj,kk+1).a_in(aa);
        if ii < m then mtx(ii+1,jj,kk).b_in(bb);
        if jj > 0 then mtx.(ii,jj-1,kk).c_out(cc) else cc := 0;
        cc := cc + aa*bb;
        accept c_out(z) do z := cc; end c_out;
end cell;
```

The very simplicity of the partial correctness proof shows that proving deadlock freedom and progress is relatively much more important for understanding tasking programs than termination is for sequential programs. In particular, in a complex system like this matrix multiplication package, a proof of freedom from deadlock seems essential to produce a conviction that it really does work.

The following is a sketch of a proof deadlock freedom, namely that the matrix multiplication system is not blocked unless `multiply_input` is waiting for input or `multiply_output` is waiting for its output to be accepted.

Order locations of `cell` by $\prec$: $\alpha \prec \beta$ iff α precedes β in the code. Verify `matrix` with the following cooperation invariant.

$$\begin{array}{l}
\forall(i,j,k) \in (1..m, 1..n, 1..k) \\
\quad 1 < i \leq m \wedge \mathit{location}(\mathit{mtx}(i,j,k)) \prec \texttt{a\_in} \leftrightarrow \mathit{location}(\mathit{mtx}(i-1,j,k)) \prec \texttt{a\_out} \wedge \\
\quad 1 < j \leq n \wedge \mathit{location}(\mathit{mtx}(i,j,k)) \prec \texttt{b\_in} \leftrightarrow \mathit{location}(\mathit{mtx}(i,j-1,k)) \prec \texttt{b\_out} \wedge \\
\quad 1 < j \leq p \wedge \mathit{location}(\mathit{mtx}(i,j,k)) \prec \texttt{c\_in} \leftrightarrow \mathit{location}(\mathit{mtx}(i,j,k)) \prec \texttt{c\_out}
\end{array}$$

Now once `mtx(i,j,k)` is initialized, it can be blocked only if it is waiting at one of `a_in`, `b_in`, `c_in` (only if $k > 1$), `a_out` (only if $i < m$), `b_out` (only if $j < n$), `c_in`. One can prove by induction on k that if `mtx(i,j,k)` is blocked at anything but `c_out`, then for some $i' \leq i$, $j' \leq j$, and $k' \leq k$, either `mtx(i',j',1)` is blocked at `a_in` or `mtx(1,j',k')` is blocked at `b_in`. These conditions are possible only if `multiply_input` is blocked at input. Otherwise, all tasks `mtx(i,j,p)` must be blocked at `c_out`. That can happen only if `multiply_result` is blocked at `result`.

For `m`, `n`, and `p` fixed finite numbers, finite state analysis (i.e., model checking) would be able to prove deadlock freedom, but not with `m`, `n`, and `p` unspecified positive integers.